Memory Delegation*

Kai-Min Chung**[1], Yael Tauman Kalai[2], Feng-Hao Liu[3], and Ran Raz[4]

[1] Department of Computer Science, Cornell University, Ithaca, NY, USA.
chung@cs.cornell.edu
[2] Microsoft Research New England, Cambridge MA, USA, yael@microsoft.com
[3] Department of Computer Science, Brown University, Providence RI, USA.
fenghao@cs.brown.edu
[4] Department of Mathematics and Computer Science, Weizmann Institute of Science, Rehovot, Israel. ran.raz@weizmann.ac.il

Abstract. We consider the problem of delegating computation, where the delegator doesn't even know the input to the function being delegated, and runs in time significantly smaller than the input length.
For example, consider the setting of *memory delegation*, where a delegator wishes to delegate her entire memory to the cloud. The delegator may want the cloud to compute functions on this memory, and prove that the functions were computed correctly. As another example, consider the setting of *streaming delegation*, where a stream of data goes by, and a delegator, who cannot store this data, delegates this task to the cloud. Later the delegator may ask the cloud to compute statistics on this streaming data, and prove the correctness of the computation. We note that in both settings the delegator must keep a (short) certificate of the data being delegated, in order to later verify the correctness of the computations. Moreover, in the streaming setting, this certificate should be computed in a streaming manner.
We construct both memory and streaming delegation schemes. We present non-interactive constructions based on the (standard) delegation scheme of Goldwasswer *et. al.* [GKR08]. These schemes allow the delegation of any function computable by an $\mathcal{L}$-uniform circuit of low depth (the complexity of the delegator depends linearly on the depth). For memory delegation, we rely on the existence of a polylog PIR scheme, and for streaming, we rely on the existence of a fully homomorphic encryption scheme.
We also present constructions based on the CS-proofs of Micali. These schemes allow the delegation of any function in $\mathbf{P}$. However, they are interactive (i.e., consists of 4 messages), or are non-interactive in the Random Oracle Model.

1 Introduction

The problem of delegating computation considers a scenario where one party, the *delegator*, wishes to delegate the computation of a function f to another

* A full version of this paper can be found on [CKLR11]
** Supported by US-Israel BSF grant 2006060 and NSF grant CNS-0831289.

party, the *worker*. The challenge is that the delegator may not trust the worker, and thus it is desirable to have the worker "prove" that the computation was done correctly. Obviously, verifying this proof should be easier than doing the computation.

This concept of "outsourcing" computation received a lot of attention in recent years, partly due to the increasing interest in cloud computing, where the goal is to outsource all the computational resources to a (possibly untrusted) "cloud". There are several reasons why the client (or delegator) may not trust the cloud, and thus would like to receive proofs for the correctness of the computation. For example, the cloud may have an incentive to return incorrect answers. Such an incentive may be a financial one, if the real computation requires a lot of work, whereas computing incorrect answers requires less work and is unlikely to be detected by the client. Moreover, in some cases, the applications outsourced to the cloud may be so critical that the delegator wishes to rule out accidental errors during the computation.

In order to ensure that the worker (or the cloud) performed the computation correctly, we would like the worker to *prove* this to the delegator. Of course, it is essential that the time it takes to verify the proof is significantly smaller than the time needed to actually run the computation. At the same time, the running time of the worker carrying out the proof should also be reasonable — comparable to the time it takes to do the computation.

The problem of delegating computation has been studied excessively (see Section 1.2 for an overview on previous work). However, most previous work on delegation allow the delegator to run in time polynomial in the input size, as long as this runtime is significantly smaller than the time it takes to do the computation. For example, when delegating the computation of a function f that runs in time T and has inputs of size n, typically the desired runtime of the delegator is $\mathrm{poly}(n, \log T)$ and the desired runtime of the worker is $\mathrm{poly}(T)$.

In this work, we want the delegator to run in time that is even smaller than the input size n. Namely, we don't allow the delegator even to read the input! At first, this requirement may seem unreasonable and unachievable. So, let us start by motivating this requirement with two examples.

Memory delegation. Suppose that Alice would like to store all her memory in the cloud. The size of her memory may be huge (for example, may include all the emails she ever received). Moreover, suppose she doesn't trust the cloud. Then, every time she asks the cloud to carry out some computation (for example, compute how many emails she has received from Bob during the last year), she would like the answer to be accompanied by a proof that indeed the computation was done correctly. Note that the input to these delegated functions may be her entire memory, which can be huge. Therefore, it is highly undesirable that Alice runs in time that is proportional to this input size. More importantly, Alice doesn't even hold on to this memory anymore, since she delegated it to the cloud.

Thus, in a memory delegation scheme, a delegator delegates her entire memory to the cloud, and then may ask the could to compute functions of this

memory, and expects the answers to be accompanied by a proof. Note that in order to verify the correctness of these proofs, the delegator must save some short certificate of her memory, say a certificate of size $\text{polylog}(n)$, where n is the memory size. The proofs should be verifiable very efficiently; say, in time $\text{polylog}(n, T)$, where T is the time it takes to compute the function. Moreover, Alice should be able to update her memory efficiently.

Streaming delegation. Suppose that there is some large amount of data that is streaming by, and suppose that a user, Alice, wishes to save this data, so that later on she will be able to compute statistics on this data. However, Alice's memory is bounded and she cannot store this data. Instead, she wishes to delegate this to the cloud. Namely, she asks the cloud to store this streaming data for her, and then she asks the cloud to perform computation on this data. As in the case of memory delegation, in order to later verify the correctness of these computations, Alice must save some short certificate of this streaming data. As opposed to the setting of memory delegation, here the certificate should be computed (and updated) in a streaming manner.

The settings of memory delegation and streaming delegation are quite similar. In both settings Alice asks the cloud to store a huge object (either her memory or the streaming data). There are two main differences between the two: (1) In the setting of streaming delegation, the certificates and updates must be computed in a streaming manner. Thus, in this sense, constructing streaming delegation schemes may be harder than constructing memory delegation schemes. Indeed, our streaming delegation scheme is more complicated than our memory delegation scheme, and proving soundness in the streaming setting is significantly harder than proving soundness in the memory setting. (2) In the setting of streaming delegation, the memory is updated by simply adding elements to it. This is in contrast to the setting of memory delegation, where the memory can be updated in arbitrary ways, depending on the user's needs. However, in the memory setting, we allow the delegator to use the help of the worker when updating her certificate (or secret state), whereas in the streaming setting we require that the delegator updates her certificate on her own. The reason for this discrepancy, is that in the memory setting the delegator may not be able to update her certificate on her own, since she may want to update her memory in involved ways (such as, erase all emails from Bob). On the other hand, in the streaming setting, it seems essential that the delegator updates her certificate on her own, since in this setting the data may be streaming by very quickly, and there may not be enough time for the delegator and worker to interact during each update.

1.1 Our Results

We construct both memory delegation and streaming delegation schemes. The memory delegation scheme consists of an offline phase, where the delegator D delegates her memory $x \in \{0,1\}^n$ to a worker W. This phase is non-interactive,

where the delegator sends a single message, which includes her memory content x to the worker W. The runtime of both the delegator and the worker in the offline phase is $\text{poly}(n)$, where n is the memory size. At the end of this phase, the delegator saves a short certificate σ of her memory, which she will later use when verifying delegation proofs.

The streaming delegation scheme, on the other hand, doesn't have such an offline phase. In the streaming setting, we consider the scenario where at each time unit t a bit x_t is being streamed. The delegator starts with some secret state (or certificate) σ_0, and at time unit $t+1$ she uses her secret state σ_t and the current bit x_{t+1} being streamed, to efficiently update her secret state from σ_t to σ_{t+1}.

In both settings, each time the delegator D wants the worker W to compute a function $f(x)$, they run a delegation protocol, which we denote by $\mathsf{Compute}(f)$. The memory delegation scheme also has an Update protocol, where the delegator D asks the worker W to update her memory and to help her update her secret state σ. The latter can be thought of as a delegation request, and the guarantees (in term of runtime and communication complexity) are similar to the guarantees of the $\mathsf{Compute}$ protocol.

In the streaming setting, the delegator updates her secret state on her own in time $\text{polylog}(N)$, where N is an upper bound on the length of the stream. Namely, the update function, that takes as input a certificate σ_t and a bit x_{t+1}, and outputs a new certificate σ_{t+1}, can be computed in time $\text{polylog}(N)$.

We present two memory and streaming delegation protocols. The first are non-interactive (i.e, $\mathsf{Compute}(f)$ consists of two messages, the first sent by the delegator and the second sent by the worker). They are based on the non-interactive version of the delegation protocol of Goldwasser *et. al.* [GKR08,KR09], denoted by GKR (though are significantly more complicated than merely running GKR). As in GKR, the efficiency of the delegator depends linearly on the depth of the circuit being delegated. Our second memory and streaming delegation protocols are interactive (i.e., $\mathsf{Compute}(f)$ consists of four messages). These schemes are based on CS-proofs of Micali [Mic94], and allow for efficient delegation of all functions in $\mathbf{P}$.

In what follows, we state our theorems formally. However, due to the lack of space, we refer the reader to the full version of this paper [CKLR11] for the formal definition of a memory delegation scheme and a streaming delegation scheme.

Theorem 1 (Memory Delegation). *Assume the existence of a poly-log PIR scheme, and assume the existence of a collision resistant hash family. Let $\mathcal{F}$ be the class of all $\mathcal{L}$-uniform poly-size boolean circuits. Then there exists a non-interactive (2-message) memory delegation scheme* mDel*, for delegating any function $f \in \mathcal{F}$. The delegation scheme,* mDel *has the following properties, for security parameter k.*

- *The scheme has perfect completeness and negligible (reusable) soundness error.*

– *The delegator and worker are efficient in the offline stage; i.e., both the delegator and the worker run in time* $\mathrm{poly}(k, n)$.
– *The worker is efficient in the online phase. More specifically, it runs in time* $\mathrm{poly}(k, S)$ *during each* $\mathsf{Compute}(f)$ *and* $\mathsf{Update}(f)$ *operation, where* S *is the size of the* $\mathcal{L}$*-uniform circuit computing* f*. The delegator runs in time* $\mathrm{poly}(k, d)$ *during each* $\mathsf{Compute}(f)$ *and* $\mathsf{Update}(f)$ *operation, where* d *is the depth of the* $\mathcal{L}$*-uniform circuit computing* f*.*[5]

In particular, assuming the existence of a poly-logarithmic PIR scheme, and assuming the existence of a collision resistent hash family, we obtain a memory delegation scheme for $\mathcal{L}$-uniform **NC** computations, where the delegator D runs in time *poly-logarithmic* in the length of the memory.

Theorem 2 (Streaming Delegation). *Let* k *be a security parameter, and let* N *be a parameter (an upper bound on the length of the stream). Let* $\mathcal{F}$ *be the class of all* $\mathcal{L}$*-uniform poly-size boolean circuits. Assume the existence of a fully-homomorphic encryption scheme secure against* $\mathrm{poly}(N)$*-size adversaries. Then there exists a streaming delegation scheme* $\mathsf{sDel}_{\mathcal{F}}$ *for* $\mathcal{F}$ *with the following properties.*

– $\mathsf{sDel}_{\mathcal{F}}$ *has perfect completeness and negligible reusable soundness error.*
– D *updates her secret state in time* $\mathrm{polylog}(N)$*, per data item.*
– *In the delegation protocol, when delegating a function* $f \in \mathcal{F}$ *computable by an* $\mathcal{L}$*-uniform circuit of size* S *and depth* d*, the delegator* D *runs in time* $\mathrm{poly}(k, d, \log N)$*, and the worker* W *runs in time* $\mathrm{poly}(k, \log N, S)$*.*

In particular, assuming the existence of a fully-homomorphic encryption scheme secure against adversaries of size $\mathrm{poly}(N)$, we obtain a streaming delegation scheme for $\mathcal{L}$-uniform **NC** computations, where the delegator D runs in time *poly-logarithmic* in the length of data stream.

Remark. We note that the property we needed from the GKR protocol is that the verifier does not need to read the entire input in order to verify, but rather only needs to access a single random point in the low-degree extension of the input. (We refer the reader to Section 2.1 for the definition and properties of a low-degree extension.) We note that the CS-proof delegation scheme of Micali [Mic94], for delegating the computation of (uniform) Turing machines, also has the property that verification can be done by only accessing a few random points in the low-degree extension of the input, assuming the underlying PCP is a PCP of Proximity [BSGH+05].

Indeed using this delegation scheme, we get a memory delegation scheme and a streaming delegation scheme for all of **P**. Using this scheme, the $\mathsf{Compute}(f)$ protocol is interactive (i.e., it is a 4-message protocol). The runtime of the delegator is $\mathrm{polylog}(T)$ and the runtime of the worker is $\mathrm{poly}(T)$, where T is the

[5] Thus, for every constant $c \in \mathbb{N}$, if we restrict the depth of f to be at most k^c, then the delegator is considered efficient.

runtime of the Turing machine computing the function f.[6] Furthermore, the memory delegation scheme relies only on the existence of a collision resistant hash family, without the need of a poly-log PIR scheme.

Theorem 3 (Interactive Memory Delegation). *Assume the existence of a collision resistant hash family. Then there exists a memory delegation scheme* mDel*, for delegating any function computable by a polynomial-time Turning machine. The delegation scheme,* mDel *has the following properties, for security parameter* k*.*

- *The scheme has perfect completeness and negligible (reusable) soundness error.*
- *The delegator and worker are efficient in the offline stage; i.e., both the delegator and the worker run in time* $\mathrm{poly}(k, n)$*.*
- *The worker is efficient in the online phase. More specifically, it runs in time* $\mathrm{poly}(k, T)$ *during each* $\mathsf{Compute}(f)$ *and* $\mathsf{Update}(f)$ *operation, where* T *is an upper-bound on the running time of* f*. The delegator runs in time* $\mathrm{poly}(k, \log T)$ *during each* $\mathsf{Compute}(f)$ *and* $\mathsf{Update}(f)$ *operation.*
- *Both* $\mathsf{Compute}(f)$ *and* $\mathsf{Update}(f)$ *operations consist of* 4 *message exchanges.*

Theorem 4 (Interactive Streaming Delegation). *Let* k *be a security parameter, and let* N *be a parameter (an upper bound on the length of the stream). Let* $\mathcal{F}$ *be the class of all functions computable by a polynomial-time Turning machine. Assume the existence of a fully-homomorphic encryption scheme secure against* $\mathrm{poly}(N)$*-size adversaries. Then there exists a streaming delegation scheme* $\mathsf{sDel}_{\mathcal{F}}$ *for* $\mathcal{F}$ *with the following properties.*

- $\mathsf{sDel}_{\mathcal{F}}$ *has perfect completeness and negligible reusable soundness error.*
- D *updates her secret state in time* $\mathrm{polylog}(N)$*, per data item.*
- *In the delegation protocol, when delegating a function* $f \in \mathcal{F}$ *computable in time* T*, the delegator* D *runs in time* $\mathrm{poly}(k, \log N, \log T)$*, and the worker* W *runs in time* $\mathrm{poly}(k, \log N, T)$*. The delegation protocol consists of* 4 *message exchanges.*

We note that in the Random Oracle Model (ROM) [BR97], the delegation scheme of Micali is non-interactive. This yields a non-interactive memory delegation scheme and a non-interactive streaming delegation scheme, for delegating all functions in $\mathbf{P}$, in the ROM.

Due to the lack of space, we focus on our results using the GKR delegation protocol, and refer the reader to the full version of this paper [CKLR11] for the details on our results using the CS-delegation protocol. However, we note that the techniques and proofs are essentially the same in both cases.

[6] We assume that $T \geq n$.

1.2 Previous Work

Various delegation protocols have been proposed in the literature. Some provide delegation protocols that are sound against any cheating worker, whereas others provide delegation protocols that are secure only against computationally bounded cheating worker (i.e., arguments as opposed to proofs). Some of these protocols are interactive, whereas others are non-interactive. We survey some of these results below, however, we emphasize that in all these solutions, the delegator runs in time that is (at least) linear in the input size, and thus do not apply to our settings of memory delegation or streaming delegation.

Interactive proofs. The celebrated IP=PSPACE Theorem [LFKN92,Sha92] yields interactive proofs for any function f computable in polynomial space, with a verifier (delegator) running in polynomial time. Thus, the IP=PSPACE protocol can be seen as a delegation protocol for languages in PSPACE $\setminus$ **P**. However, the complexity of the prover (worker) is only bounded by polynomial space (and hence exponential time). This theorem was refined and scaled down in [FL93] to give verifier complexity $\text{poly}(n, s)$ and prover complexity $2^{\text{poly}(s)}$ for functions f computable in time T and space s, on inputs of length n. Note that the prover complexity is still super-polynomial in T, even for computations that run in the smallest possible space, namely $s = O(\log T)$.

The prover complexity was recently improved by Goldwasser et al. [GKR08] to $\text{poly}(T, 2^s)$, which is $\text{poly}(T)$ when $s = O(\log T)$. More generally, Goldwasser et al. [GKR08] give interactive proofs for computations of small *depth* d (i.e. parallel time). For these, they achieve prover complexity $\text{poly}(T)$ and verifier complexity $\text{poly}(n, d, \log T)$. (This implies the result for space-bounded computation because an algorithm that runs in time T and space s can be converted into one that runs in time $\text{poly}(T, 2^s)$ and depth $d = O(s^2)$.) However, if we do not restrict to computations of small space or depth, then we cannot use interactive proofs. Indeed, any language that has an interactive proof with verifier running time (and hence communication) T_V can be decided in space $\text{poly}(n, T_V)$.

Interactive arguments. Interactive arguments [BCC88] (aka computationally sound proofs [Mic00]) relax the soundness condition to be computational. Namely, instead of requiring that no prover strategy whatsoever can convince the verifier of a false statement, we instead require that no computationally feasible prover strategy can convince the verifier of a false statement. In this model, Kilian [Kil92] and Micali [Mic00] gave constant-round protocols with prover complexity $\text{poly}(T, k)$ and verifier complexity $\text{poly}(n, k, \log T)$ (where k is the security parameter), assuming the existence of collision-resistant hash functions [BG02].

Toward non-interactive Solutions. This possibility of efficient non-interactive arguments was suggested by Micali [Mic00], who showed that non-interactive arguments with prover complexity $\text{poly}(T, k)$ and verifier complexity $\text{poly}(n, k, \log T)$

are possible in the Random Oracle Model (the oracle is used to eliminate interaction a la Fiat–Shamir [FS86]). Heuristically, one might hope that by instantiating the random oracle with an appropriate family of hash functions, we could obtain a non-interactive solution to delegating computation: first the delegator (or a trusted third party) chooses and publishes a random hash function from the family, and then, the proofs are completely non-interactive (just one message from the prover to the verifier). However, the Random Oracle Heuristic is known to be unsound in general [CGH04] and even in the context of Fiat–Shamir [Bar01,GK03]. Thus, despite extensive effort, the existence of efficient non-interactive arguments remains a significant open problem in complexity and cryptography.

There has been some recent progress in reducing the amount of interaction needed. Using a transformation of Kalai and Raz [KR09], the GKR delegation protocol [GKR08] can be converted into a 2-message argument (assuming the existence of single-server private-information retrieval (PIR) schemes). However, like the interactive proofs of [GKR08], this solution applies only to small-depth computations, as the verifier's complexity grows linearly with the depth.

Very recently, Gennaro, Gentry, and Parno [GGP10], and the followup work of Chung, Kalai, and Vadhan [CKV10], gave a 2-message delegation scheme for arbitrary functions. However, these constructions have an offline phase, where the delegator invests time $\text{poly}(T, k)$ and computes a *secret state* (T is the time it takes to compute the function, and k is the security parameter). In the online phase, the delegator's running time is reduced to $\text{poly}(n, k, \log T)$ for an input of length n, and the worker's complexity is $\text{poly}(T, k)$. Thus, the delegator's large investment in the offline phase can be amortized over many executions of the online phase to delegate the computation of f on many inputs. Their online phase is not completely non-interactive, but rather consists of two messages. However, in many applications, two messages will be necessary anyway, as the delegator may need to communicate the input x to the worker.

We remark that one main drawback of these works [GGP10,CKV10] is that soundness is only guaranteed as long as the adversarial worker does not learn whether the delegator accepted or rejected the proofs.

In another followup work, Applebaum, Ishai, and Kushilevitz [AIK10] also consider the offline/online setting, but focus on efficient solutions for one-time delegation (i.e., the online phase can only be executed one time). They also consider the case when the delegation functions are represented as arithmetic circuits.

PCPs and MIPs. The MIP=NEXP Theorem [BFL91] and its scaled-down version by Babai et al. [BFLS91] yield multi-prover interactive proofs and probabilistically checkable proofs for time T computations with a prover running in time $\text{poly}(T)$ and a verifier running in time $\text{poly}(n, \log T)$, exactly as we want. However, using these for delegation require specialized communication models — either 2 non-communicating provers, or a mechanism for the prover to give the verifier random access to a long PCP (of length $\text{poly}(T)$) that cannot be changed by the prover during the verification.

Streaming Interactive Proofs. Recently, Cormode, Thaler, and Yi [CTY10] considered streaming interactive proofs, which is a strengthening of interactive proofs where the input is given to the verifier in a streaming manner and the verifier is restricted to have sub-linear (ideally, poly-logarithmic) space. They observed that both the GKR protocol [GKR08] and universal arguments [BG02] can be modified to yield efficient streaming interactive proofs/arguments.

Streaming interactive proofs are closely related to streaming delegation. The main difference is that streaming interactive proofs correspond to *one-time* streaming delegation, whereas in our streaming delegation model, the delegator is allowed to delegate as many computations to the worker as she want. Indeed, the GKR protocol is also the starting point of our construction of streaming delegation scheme, and the main effort is to make the scheme *reusable*.

2 Preliminaries

2.1 Low Degree Extension

Let $\mathbb{H}$ be an extension field of $\mathbb{GF}[2]$, and let $\mathbb{F}$ be an extension field of $\mathbb{H}$ (and in particular, an extension field of $\mathbb{GF}[2]$), where $|\mathbb{F}| = \text{poly}(|\mathbb{H}|)$.[7] We always assume that field operations can be performed in time that is poly-logarithmic in the field size. Fix an integer $m \in \mathbb{N}$. In what follows, we define the low degree extension of an n-element string $(w_0, w_1, \ldots, w_{n-1}) \in \mathbb{F}^n$ with respect to $\mathbb{F}, \mathbb{H}, m$, where $n \leq |\mathbb{H}|^m$.

Fix $\alpha : \mathbb{H}^m \to \{0, 1, \ldots, |\mathbb{H}|^m - 1\}$ to be any (efficiently computable) one-to-one function. In this paper, we take α to be the lexicographic order of $\mathbb{H}^m$. We can view $(w_0, w_1, \ldots, w_{n-1})$ as a function $W : \mathbb{H}^m \to \mathbb{F}$, where

$$W(z) = \begin{cases} w_{\alpha(z)} & \text{if } \alpha(z) < n, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

A basic fact is that there exists a unique extension of W into a function $\tilde{W} : \mathbb{F}^m \to \mathbb{F}$ (which agrees with W on $\mathbb{H}^m$; i.e., $\tilde{W}|_{\mathbb{H}^m} \equiv W$), such that $\tilde{W}$ is an m-variate polynomial of degree at most $|\mathbb{H}| - 1$ in each variable. Moreover, as is formally stated in the proposition below, the function $\tilde{W}$ can be expressed as

$$\tilde{W}(t_1, \ldots, t_m) = \sum_{i=0}^{n-1} \tilde{\beta}_i(t_1, \ldots, t_m) \cdot w_i,$$

where each $\tilde{\beta}_i : \mathbb{F}^m \to \mathbb{F}$ is an m-variate polynomial, that depends only on the parameters $\mathbb{H}, \mathbb{F}$, and m (and is independent of w), of size $\text{poly}(\mathbb{H}, m)$ and degree $|\mathbb{H}| - 1$ in each variable.

[7] Usually, when doing low degree extensions, $\mathbb{F}$ is taken to be an extension field of $\mathbb{GF}[2]$, and $\mathbb{H}$ is simply a subset of $\mathbb{F}$ (not necessarily a subfield). In this work, following the work of [GKR08], we take $\mathbb{H}$ to be a subfield. However, all that is actually needed is that it is of size 2^ℓ for some $\ell \in \mathbb{N}$.

The function $\tilde{W}$ is called the *low degree extension* of $w = (w_0, w_1, \ldots, w_{n-1})$ with respect to $\mathbb{F}, \mathbb{H}, m$, and is denoted by $\mathrm{LDE}_w^{\mathbb{F},\mathbb{H},m}$. We omit the index of $\mathbb{F}, \mathbb{H}, m$ when the context is clear. Also, sometimes we use $\tilde{W}$ for simplicity.

Proposition 1. *There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\mathbb{GF}[2]$,[8] an extension field $\mathbb{F}$ of $\mathbb{H}$, and integer m. The machine runs in time $\mathrm{poly}(|\mathbb{H}|, m)$ and outputs the unique $2m$-variate polynomial $\tilde{\beta} : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ of degree $|\mathbb{H}|-1$ in each variable (represented as an arithmetic circuit of degree $|\mathbb{H}|-1$ in each variable), such that for every $w = (w_0, w_1, \ldots, w_{n-1}) \in \mathbb{F}^n$, where $n \leq |\mathbb{H}|^m$, and for every $z \in \mathbb{F}^m$,*

$$\tilde{W}(z) = \sum_{p \in \mathbb{H}^m} \tilde{\beta}(z, p) \cdot W(p),$$

where $W : \mathbb{H}^m \to \mathbb{F}$ is the function corresponding to $(w_0, w-1, \ldots, w_{n-1})$ as defined in Equation (1), and $\tilde{W} : \mathbb{F}^m \to \mathbb{F}$ is its low degree extension (i.e., the unique extension of $W : \mathbb{H}^m \to \mathbb{F}$ of degree at most $\mathbb{H}-1$ in each variable).

Moreover, $\tilde{\beta}$ can be evaluated in time $\mathrm{poly}(|\mathbb{H}|, m)$. Namely, there exists a Turing machine that runs in time $\mathrm{poly}(|\mathbb{H}|, m)$ that takes as input parameters $\mathbb{H}, \mathbb{F}, m$ (as above), and a pair $(z, p) \in \mathbb{F}^m \times \mathbb{F}^m$, and outputs $\tilde{\beta}(z, p)$.

Corollary 1. *There exists a Turing machine that takes as input an extension field $\mathbb{H}$ of $\mathbb{GF}[2]$, an extension field $\mathbb{F}$ of $\mathbb{H}$, an integer m, a sequence $w = (w_0, w_1, \ldots, w_{n-1}) \in \mathbb{F}^n$ such that $n \leq |\mathbb{H}|^m$, and a coordinate $z \in \mathbb{F}^m$. It runs in time $n \cdot \mathrm{poly}(|\mathbb{H}|, m)$, and outputs the value $\tilde{W}(z)$, where $\tilde{W}$ is the unique low-degree extension of w (with respect to $\mathbb{H}, \mathbb{F}, m$).*

2.2 Delegation Schemes

In recent years, as cloud computing is gaining popularity, there have been many attempts to construct efficient delegation schemes. Loosely speaking, a delegation scheme is a protocol between a delegator D and a worker W, where the delegator asks the worker to do some computation, and prove that he indeed did the computation correctly. Typically, a delegation scheme is with respect to a class of functions $\mathcal{F}$, and the requirement is that on input (f, x) where $f \in \mathcal{F}$ and x is in the domain of f, the worker outputs $f(x)$, along with a proof (which may be interactive or non-interactive). The requirement is that the worker runs in time that is polynomial in the size of f (when representing f as a circuit), and the delegator runs in time that is significantly shorter than the size of f (as otherwise, it would simply compute $f(x)$ on its own). In this work, we use the 2-message delegation protocol of [GKR08], which in turn uses a round reduction technique from [KR09]. The protocol has the following guarantees.

[8] Throughout this work, when we refer to a machine that takes as input a field, we mean that the machine is given a short (poly-logarithmic in the field size) description of the field, that permits field operations to be computed in time that is poly-logarithmic in the field size.

Theorem 5. *[GKR08,KR09] Assume the existence of a poly-logarithmic PIR scheme. Let k be the security parameter, and let $\mathcal{F}$ be the family of functions computable by $\mathcal{L}$-space uniform boolean circuits of size $S(n)$ and depth $d(n) \geq \log S(n)$. Then, there exists a delegation protocol for $\mathcal{F}$ with the following properties.*

1. *The worker runs in time* $\mathrm{poly}(S,k)$ *and the delegator runs in time* $n \cdot \mathrm{poly}(k, d(n))$.
2. *The protocol has perfect completeness and soundness $s \leq \frac{1}{2}$ (can be made arbitrarily small), where soundness is against any cheating worker of size $\leq 2^{k^3}$.*
3. *The protocol consists of two messages, with communication complexity* $d(n) \cdot \mathrm{poly}(k, \log S)$. *Moreover, the first message sent by the delegator depends only on her random coin tosses, and is independent of the statement being proved.*
4. *If the delegator is given oracle access to the low-degree extension of x, rather than being given the input x itself, then it runs in time* $\mathrm{poly}(k, d(n))$, *and the protocol still has all the properties described above, assuming the parameters $\mathbb{H}, \mathbb{F}, m$ of the low-degree extension satisfy the following:*

$$|\mathbb{H}| = \theta(d \cdot \log n),\ m = \theta\left(\frac{\log n}{\log d}\right),\ |\mathbb{F}| = \mathrm{poly}(|\mathbb{H}|)$$

where poly *is a large enough polynomial.*[9] *Moreover, the delegator queries the low-degree extension of x at a single point, which is uniformly random (over his coin tosses).*

Throughout this paper, we denote this protocol by GKR.

2.3 Merkle Tree Commitments

Definition 1. *Let $h : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ be a hash function. A Merkle tree commitment of a sting $x \in \{0,1\}^n$ w.r.t. h, denoted by $T_h(x)$, is a k-bit string, computed as follows: The input x is partitioned into $m = \lceil n/k \rceil$ blocks $x = (B_1, \ldots, B_m)$, each block of size k. These blocks are partitioned into pairs (B_{2i-1}, B_{2i}), and the hash function h is applied to each pair, resulting in $m/2$ blocks. Then, again these $m/2$ blocks are partitioned into pairs, and the hash function h is applied to each of these pairs, resulting with $m/4$ blocks. This is repeated* $\log m$ *times, resulting in a binary tree of hash values, until one block remains. This block is $T_h(x)$.*

3 Overview of Our Constructions

In what follows we present a high-level overview of our memory and streaming delegation schemes. In this extended abstract, we focus on our non-interactive constructions, based on the GKR delegation schemes, and only present the high-level overview of these constructions. We refer the reader to the full version of this paper [CKLR11] for a formal presentation of our constructions and analysis.

[9] The larger poly is, the smaller the soundness becomes.

3.1 Overview of our Memory Delegation Scheme

The starting point of this work is the observation of Goldwasswer *et. al.* [GKR08], that their delegation protocol can be verified *very* efficiently (in time sub-linear in the input size), if the delegator has oracle access to the low-degree extension of the input x. Moreover, as observed by [GKR08], the delegator needs to access this low-degree extension LDE_x at a single point z, which depends only on the random coin tosses of the delegator.

This observation immediately gives rise to a memory delegation scheme with *one-time* soundness: The delegator's secret state will be $(z, \mathrm{LDE}_x(z))$. Then, she will use this secret state in order to verify computation using the GKR protocol. As was argued by Goldwasswer *et. al.*, this indeed works if the delegator runs the delegation protocol *once*. However, the soundness crucially relies on the fact that the delegator's secret state is indeed secret, and if the delegator uses this state more than once, then soundness breaks completely.

One idea, following the idea of Gennaro *et. al.* [GGP10], is to use a fully homomorphic encryption (FHE) scheme to encrypt all the communication, in order to hide the secret state. This indeed works if the worker does not learn whether the delegator accepts or rejects his proofs. However, if the worker does learn the verdict of the delegator, then there are known attacks that break soundness.

In the streaming setting, we follow this approach, and we succeed in overcoming this problem, and construct a scheme that is sound even if the worker does learn the verdict of the delegator. We could follow this approach in the memory delegation setting as well. However, for several reasons, we choose to take a different approach. First, the approach above relies on the existence of an FHE scheme, whereas our memory delegation scheme relies on the existence of a poly-logarithmic PIR scheme, arguably a more reasonable assumption. Second, the approach above results with the delegator having a secret state, whereas in our memory delegation scheme, the state of the delegator is public. Finally, the construction and proof of the memory delegation scheme is simpler.

In our approach, instead of having $(z, \mathrm{LDE}_x(z))$ as the delegator's secret state, the delegator keeps a tree-commitment of the entire LDE_x as her secret state (see Section 2.3 for the definition of a tree-commitment). Namely, she chooses a random hash function h from a collision-resistant hash family, and keeps $(h, T_h(\mathrm{LDE}_x))$ as her state. In addition to giving the worker her memory x, she also gives him the hash function h. We stress that her state is not secret, which makes the proof of security significantly simpler than that in the streaming setting (where the delegator's state is secret).

Very roughly speaking, when the delegator wishes to delegate the computation of a function f, they execute $\mathsf{Compute}(f)$ by simply running the (non-interactive) delegation protocol $\mathrm{GKR}(f)$. Recall that at the end of the GKR protocol the delegator needs to verify the value of $\mathrm{LDE}_x(r)$ for a random r. However, she doesn't have x, since it was delegated to the prover, and all she has is the state $(h, T_h(\mathrm{LDE}_x))$. So, rather than computing the value of $\mathrm{LDE}_x(r)$

on her own, the worker will reveal this value, by sending the augmented path in the Merkle tree corresponding to the leaf r.

Unfortunately the high-level description given above is a gross oversimplification of our actual scheme, and there are several technical issues that complicate matters. We elaborate on these in Section 3.3.

We mention that when the delegator wishes to update her memory from x to $g(x)$, she needs to update her secret state from $(h, T_h(\mathrm{LDE}_x))$ to $(h, T_h(\mathrm{LDE}_{g(x)}))$.[10] However, she cannot perform this operation on her own, since she does not have x. Instead she will delegate this computation to the worker, by requesting a $\mathsf{Compute}(g')$ operation, where $g'(x) = T_h(\mathrm{LDE}_{g(x)})$.

3.2 Overview of our Streaming Delegation Scheme

Our streaming delegation scheme is similar to our memory delegation scheme described above, and the main difference is in the way the certificate is generated and updated, and in the way the worker reveals the value $\mathrm{LDE}_x(r)$.

Generating and updating the certificate. Recall that in the memory delegation scheme, the certificate of the delegator D consists of a tree-commitment to the low-degree extension of her memory x. Namely, her certificate is $(h, T_h(\mathrm{LDE}_x))$, where h is a collision resistant hash function. Note that this certificate cannot be updated in a streaming manner, since any change to x changes the low-degree extension LDE_x almost everywhere.

Instead, in the streaming setting, we replace the tree commitment with an "*algebraic commitment*", which has the property that it can be updated efficiently when new data items arrive. The resulting certificate is a random point in the low-degree extension of the stream x; i.e., $(z, \mathrm{LDE}_x(z))$ for a random point z. This certificate is efficiently updatable, if we assume some upper-bound N on the size of the stream, and we take parameters $\mathbb{H}, \mathbb{F}, m$ of the low-degree extension, such that

$$|\mathbb{H}| = \mathrm{polylog}(N),\ m = \theta\left(\frac{\log N}{\log\log N}\right),\ |\mathbb{F}| = \mathrm{poly}(|\mathbb{H}|) \tag{2}$$

(this follows from Proposition 1).

As in the memory delegation scheme, at the end of each delegation protocol, the delegator needs to verify the value of $\mathrm{LDE}_x(r)$ at a random point r. In the memory delegation scheme this was done using a Reveal protocol where the worker reveals the augmented path of the leaf r in the Merkle tree-commitment of LDE_x. In the streaming setting, the Reveal protocol is totally different, since the delegator cannot compute the tree-commitment of LDE_x. Unfortunately, unlike in the memory delegation scheme, in the streaming setting constructing a *reusable* and *sound* reveal protocol is highly non-trivial.

[10] Actually, for technical reasons she will need to choose a fresh hash function $h' \leftarrow \mathcal{H}$ during each Update. We discard this technical issue here.

The Reveal protocol. Our starting point is a basic reveal protocol Reveal_1 described in Figure 1. Note that the soundness of Reveal_1 relies on the secrecy of the certificate σ. Namely, assuming that W does not know the point z, it is not hard to see, by the Schwartz-Zippel Lemma, that an adversarial worker can cheat with probability at most $d/|\mathbb{F}|$, where d is the (total) degree of LDE_x.

Reveal_1 protocol: D stores a *secret* state $\sigma = (z, \text{LDE}_x(z))$, where $x \in \{0,1\}^N$ and z is a random point in $\mathbb{F}^m$, and wants to learn the value of $\text{LDE}_x(s)$ from W.

- D sends to W the line ℓ_{sz} that passes through the points s and z. More specifically, D chooses two random points $\alpha_1, \alpha_2 \leftarrow \mathbb{F}$, and defines $\ell_{s,z}$ to be the line that satisfies $\ell_{s,z}(\alpha_1) = z$ and $\ell_{s,z}(\alpha_2) = s$.
- W returns a univariate polynomial $p : \mathbb{F} \to \mathbb{F}$, which is the polynomial LDE_x restricted to the line $\ell_{s,z}$ (i.e., $p = \text{LDE}_x|_{\ell_{s,z}}$).
- D checks whether $p(\alpha_1) = \text{LDE}_x(z)$, and if so accepts the value $p(\alpha_2) = \text{LDE}_x(s)$. Otherwise, she rejects.

Fig. 1. Reveal_1 protocol

However, note that the Reveal_1 protocol is not reusable. Suppose that D uses the above reveal protocol to learn the value of LDE_x on two random points $s, s' \in \mathbb{F}^m$. From the two executions, an adversarial worker W^* receives two lines $\ell_{s,z}$ and $\ell_{s',z}$, and can learn the secret point z by taking the intersection of the two lines. Once W^* learns z, W^* can easily cheat by returning any polynomial p^* that agrees with LDE_x only on point z but disagrees on the remaining points.

As observed by Gennaro *et. al.* [GGP10], a natural way to protect the secret point z, is to run the above Reveal protocol under a fully-homomorphic encryption (FHE) scheme. Namely, D generates a pair of keys $(\mathsf{pk}, \mathsf{sk})$ for a FHE $(\text{Gen}, \text{Enc}, \text{Dec}, \text{Eval})$, and sends pk and an encrypted line $\hat{\ell}_{s,z} = \text{Enc}_{\mathsf{pk}}(\ell_{s,z})$ to W, who can compute the polynomial $p = \text{LDE}_x|_\ell$ homomorphically under the encryption. Indeed, by the semantic security of FHE, an adversarial worker W^* cannot learn any information from D's message $\hat{\ell}_{s,z}$. This indeed makes the protocol reusable provided that W^* does not learn the decision bits of D, as proved in [GGP10,CKV10].

However, since the decision bit of D can potentially contain one bit information about the secret point z, it is not clear that security holds if W^* learns these decision bits. In fact, for both of the delegation schemes of [GGP10,CKV10], which use FHE to hide the delegator D's secret state, there are known attacks that learn the whole secret state of D bit-by-bit from D's decision bits.

Fortunately, we are able to show that a variant of the Reveal_1 protocol described in Figure 2 is reusable even if W^* learns the decision bits of D. The main difference between Reveal_1 and Reveal_2 is that in Reveal_2, the delegator D uses a random *two-dimensional* affine subspace instead of a line, and uses an FHE to mask the entire protocol.

Reveal$_2$ protocol: D stores a secret state $\sigma = (z, \mathrm{LDE}_x(z))$, where $x \in \{0,1\}^N$ and z is a random point in $\mathbb{F}^m$, and wants to learn the value of $\mathrm{LDE}_x(s)$ from W.

- D does the following.
  1. Generate a pair of keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}(1^k)$ for a fully homomorphic encryption scheme FHE.
  2. Choose a random two-dimensional affine subspace $S_{s,z} \subset \mathbb{F}^m$ that contains the points s and z. More specifically, choose two random points $\alpha_1, \alpha_2 \leftarrow \mathbb{F}^2$ and let $S_{s,z} \subset \mathbb{F}^m$ be a random two-dimensional affine subspace that satisfies $S_{s,z}(\alpha_1) = z$ and $S_{s,z}(\alpha_2) = s$.
  3. Send $\hat{S}_{s,z} \leftarrow \mathrm{Enc}_{\mathsf{pk}}(S_{s,z})$ and pk to W.
- W homomorphically computes the two-variate polynomial $p = \mathrm{LDE}_x|_{S_{s,z}}$ under the FHE (denote the resulting ciphertext $\hat{p}$), and sends $\hat{p}$ to D.
- D decrypts and checks whether $p(\alpha_1) = \mathrm{LDE}_x(z)$, and if so accepts the value $p(\alpha_2) = \mathrm{LDE}_x(s)$.

Fig. 2. Protocol Reveal$_2$

We prove that no efficient adversarial W^* can learn useful information about the secret point z from the Reveal$_2$ protocol. We note that the proof of the above statement is highly non-trivial, and is one of the main technical difficulties in this work. Informally, we first prove a "leakage-resilient lemma", which claims that the ciphertext $\hat{S}_{rz}$ and the decision bit b of D (which depend on the strategy of W^*) do not give too much information about S_{rz} to W^*. In other words, the random subspace $S_{s,z}$ still has high (pseudo-)entropy from the point of view of W^*. Then we use an *information-theoretic* argument to argue that a random point z in a sufficiently random (with high entropy) subspace $S_{s,z}$ is *statistically close* to a random point in $\mathbb{F}^m$, which implies that W^* does not learn useful information about z.

The Field Size. Recall that by Schwartz-Zippel Lemma, an adversarial worker can cheat with probability at most $d/|\mathbb{F}|$, where d is the (total) degree of LDE_x. Recall that in our setting of parameters:

$$|\mathbb{H}| = \mathrm{polylog}(N),\ m = \theta\left(\frac{\log N}{\log\log N}\right),\ |\mathbb{F}| = \mathrm{poly}(|\mathbb{H}|).$$

Thus, a cheating worker can cheat (and more importantly, obtain information about the secret z) with probability $d/|\mathbb{F}| = O(1/\mathrm{polylog}(N))$, which is not low enough.

The idea is to reduce the cheating probability to negligible by simply increasing the field size to be super-polynomial. However, we cannot increase the field size in the GKR protocol, since it will increase the complexity of the worker. Instead, we use an extension field $\tilde{\mathbb{F}}$ of $\mathbb{F}$, of super-polynomial size, only in the certificate and the Reveal protocol, but run the GKR protocols as before. Namely, the secret state is $\sigma = (z, \mathrm{LDE}^{\tilde{\mathbb{F}},\mathbb{H},m}(z))$ where $z \leftarrow \tilde{\mathbb{F}}^m$, The GKR protocol is run exactly as before with the parameters $(\mathbb{H}, \mathbb{F}, m)$.

3.3 Additional Technicalities

The high-level description given above (in Sections 3.1 and 3.2) is a gross over-simplification of our actual schemes, and there are several technical issues that complicate matters.

Recall that in the overview above, we claimed that $\mathsf{Compute}(f)$ merely runs GKR, in addition to a Reveal protocol which helps the delegator verify the GKR protocol.[11] There are several technical reasons why this actually does not work. In what follows, we explain what are the main technical problems with this simple idea, and we give the highlevel idea of how to overcome these problems.

1. The first technicality (the easiest one to deal with), is that the GKR delegation scheme does not have a negligible soundness error. In our setting, especially in the setting of memory delegation, it is very important to have negligible soundness. The reason is that if the soundness is non-negligible, then a cheating worker may cheat in the update procedure of the memory delegation scheme (which is also being delegated). The problem is that if a worker cheats even once in an update procedure, all soundness guarantees are mute from that point on. So, we really need the soundness error to be negligible. In order to reduce the soundness error, we will run the GKR protocol in parallel u times (for any parameter u such that $1/2^u = \mathsf{ngl}(k)$, where k is the security parameter). We denote the u-fold parallel repetition of GKR by $\mathrm{GKR}^{(u)}$. As a result the worker will need to reveal to u random points in the low-degree extension: $\mathrm{LDE}_x(r_1), \ldots, \mathrm{LDE}_x(r_u)$.
2. The second technical point is more subtle. In the offline stage, when the delegator computes the tree commitment $T_h(\mathrm{LDE}_x)$, she needs to choose the parameters $\mathbb{H}, \mathbb{F}, m$ for the low-degree extension. The typical choice for these parameters is:

$$|\mathbb{H}| = \mathrm{polylog}(n),\ |\mathbb{F}| = \mathrm{poly}(|\mathbb{H}|),\ m = O\left(\frac{\log n}{\log\log n}\right),$$

where $n = |x|$. When delegating the computation of a function f, the worker and delegator run $\mathrm{GKR}^{(u)}(f)$ and need to verify $\mathrm{LDE}_x(r_i) = v_i$ for random points $r_1, \ldots, r_u$. However, here the parameters of the low-degree extension LDE_x depend on the depth d of the circuit computing f. Namely, looking at the parameters given in [GKR08] (see Theorem 5), the parameters of the low-degree extension are

$$|\mathbb{H}'| = \theta(d \cdot \log n),\ m' = \theta\left(\frac{\log n}{\log d}\right),\ |\mathbb{F}'| = \mathrm{poly}(|\mathbb{H}'|).$$

Therefore, the worker cannot simply execute the Reveal protocols of the memory delegation or the streaming delegation. In the memory setting, the tree commitment is w.r.t. parameters $\mathbb{H}, \mathbb{F}, m$ whereas the delegator needs

[11] The Reveal protocol in the memory setting is totally different from the Reveal protocol in the streaming setting.

to verify $\mathrm{LDE}_x^{\mathbb{F}',\mathbb{H}',m'}(r_i) = v_i$. In the streaming setting, the secret state of the delegator is $(z, \mathrm{LDE}_x^{\mathbb{F},\mathbb{H},m}(z))$, as opposed to $(z, \mathrm{LDE}_x^{\mathbb{F}',\mathbb{H}',m'}(z))$, thus the Reveal protocol described in Section 3.2 doesn't work.
We get around this technical problem by delegating the functions $g_{r_i}(x) \triangleq \mathrm{LDE}_x^{\mathbb{H}',\mathbb{F}',m'}(r_i)$. Luckily, these functions can be computed by a poly-size circuit of depth at most $\log^2 n$, assuming the delegated function f is of poly-size (see Proposition 1). We delegate the computation of each of these g_{r_i} using $\mathrm{GKR}^{(u)}$ to ensure negligible soundness. Thus, finally the worker will need to reveal to u^2 points in LDE_x (u points for each g_{r_i}).[12]

3. The final technical difficulty is that all these algorithms need to run in parallel, since we want our final delegation schemes to be non-interactive (i.e., to consist of only two messages). Typically, there is no problem in running several two-message protocols in parallel [BIN97,CHS05]. However, in our case, the delegator uses a common *secret* input in these protocols. Namely, the delegator uses secret randomness $r_1, \ldots, r_u \in (\mathbb{F}')^{m'}$ in the parallel repetition of the delegation protocol $\mathrm{GKR}(f)$ which ends with her needing to verify that $\mathrm{LDE}_x^{\mathbb{H}',\mathbb{F}',m'}(r_i) = v_i$ for every $i \in [u]$. In addition she uses these same r_i's in the delegation protocols $\mathrm{GKR}(g_{r_i})$. Moreover, at the end of each of the $\mathrm{GKR}(g_{r_i})$ protocols, the delegator needs to verify that $\mathrm{LDE}_x^{\mathbb{H},\mathbb{F},m}(z_{i,j}) = w_{i,j}$ for random points $z_{i,1}, \ldots, z_{i,u} \in \mathbb{F}^m$. Finally, they also run a reveal protocol for each $z_{i,j}$, denoted by $\mathrm{Reveal}(z_{i,j})$.
We note that the protocol $\mathrm{GKR}(f)$ (resp. $\mathrm{GKR}(g)$) is not sound if the r_i's (resp. $z_{i,j}$'s) are a priori known to the worker. To ensure that soundness still holds even if we run all these algorithms in parallel, we mask parts of the delegator's message using a PIR scheme or an FHE scheme, and then we claim that the soundness error remains negligible. To this end, we use a "parallel composition lemma", which roughly states that if a set of protocols $\Pi_1, \ldots \Pi_t$ are executed in parallel, and the verifiers use the same common private randomness p in all these protocols, then the soundness remains if the messages of the verifiers hide this common secret randomness p.

Acknowledgments

We are very grateful to Shai Halevi for collaborating with us in the initial phase of this work, and to Salil Vadhan for several helpful discussions.

References

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP (1)*, pages 152–163, 2010.

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS*, pages 106–115, 2001.

[12] We note that there are several ways to improve efficiency, such as thinking of $(g_{r_1}, \ldots, g_{r_u})$ as one function. However, for the sake of simplicity of exposition, we focus on the simplest (rather than most efficient) solution.

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[BFL91] László Babai, Lance Fortnow, and Carsten Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, pages 21–31, 1991.

[BG02] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity*, pages 194–203, 2002.

[BIN97] Mihir Bellare, Russell Impagliazzo, and Moni Naor. Does parallel repetition lower the error in computationally sound protocols? In *FOCS*, pages 374–383, 1997.

[BR97] Mihir Bellare and Phillip Rogaway. Minimizing the use of random oracles in authenticated encryption schemes. In *ICICS*, pages 1–16, 1997.

[BSGH+05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Short pcps verifiable in polylogarithmic time. In *IEEE Conference on Computational Complexity*, pages 120–134, 2005.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *Journal of the ACM*, 51(4):557–594, 2004.

[CHS05] Ran Canetti, Shai Halevi, and Michael Steiner. Hardness amplification of weakly verifiable puzzles. In *TCC*, pages 17–33, 2005.

[CKLR11] Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. Cryptology ePrint Archive, Report 2011/273, 2011. `http://eprint.iacr.org/`.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, pages 483–501, 2010.

[CTY10] G. Cormode, J. Thaler, and K. Yi. Verifying computations with streaming interactive proofs. Technical Report TR10-159, ECCC Report, 2010.

[FL93] Lance Fortnow and Carsten Lund. Interactive proof systems and alternating time-space complexity. *Theoretical Computer Science*, 113(1):55–73, 1993.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[GK03] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the fiat-shamir paradigm. pages 102–113, 2003.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *STOC*, pages 113–122, 2008.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, pages 723–732, 1992.

[KR09] Yael Tauman Kalai and Ran Raz. Probabilistically checkable arguments. In *CRYPTO*, 2009.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, 1992.

[Mic94] Silvio Micali. Cs proofs (extended abstracts). In *FOCS*, pages 436–453, 1994.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[Sha92] Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.